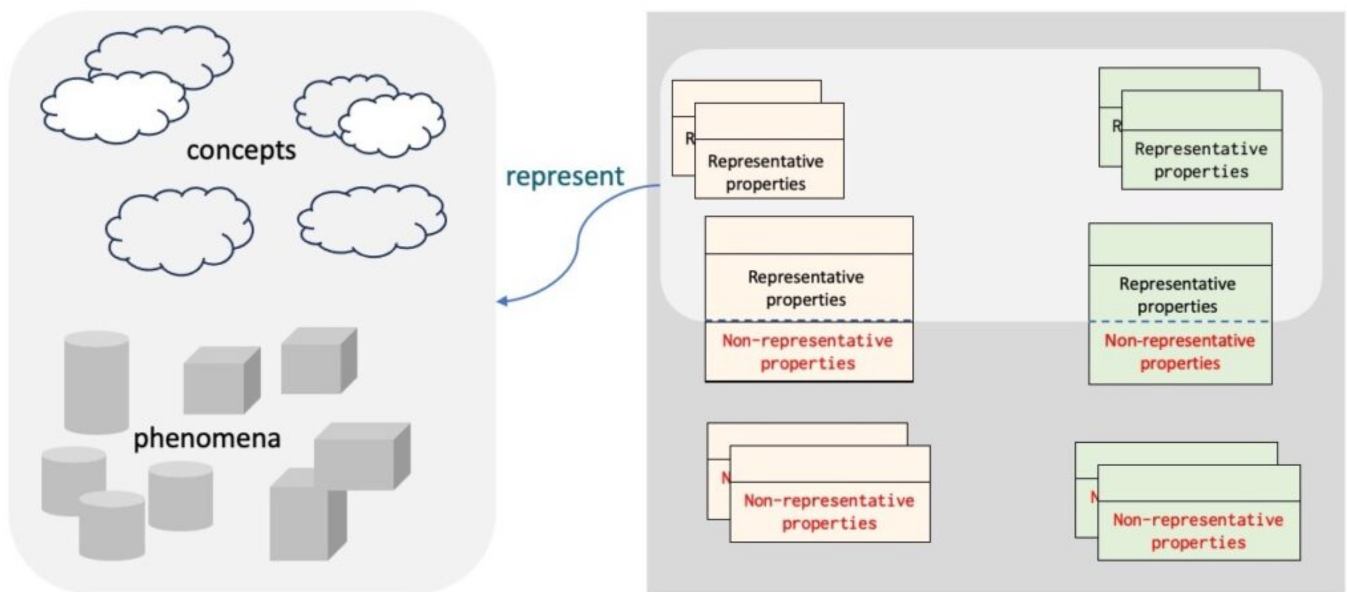
14. Representative and non-representative parts

In this chapter, we discuss representative and non-representative parts of a program with respect to modeling a given domain.

A primary focus of this book has been to emphasize that programming is modeling in the sense that programming is about creating a model of the relevant aspects of the application domain. This implies that one should attempt to represent descriptions of phenomena and their properties by corresponding language elements in the program.

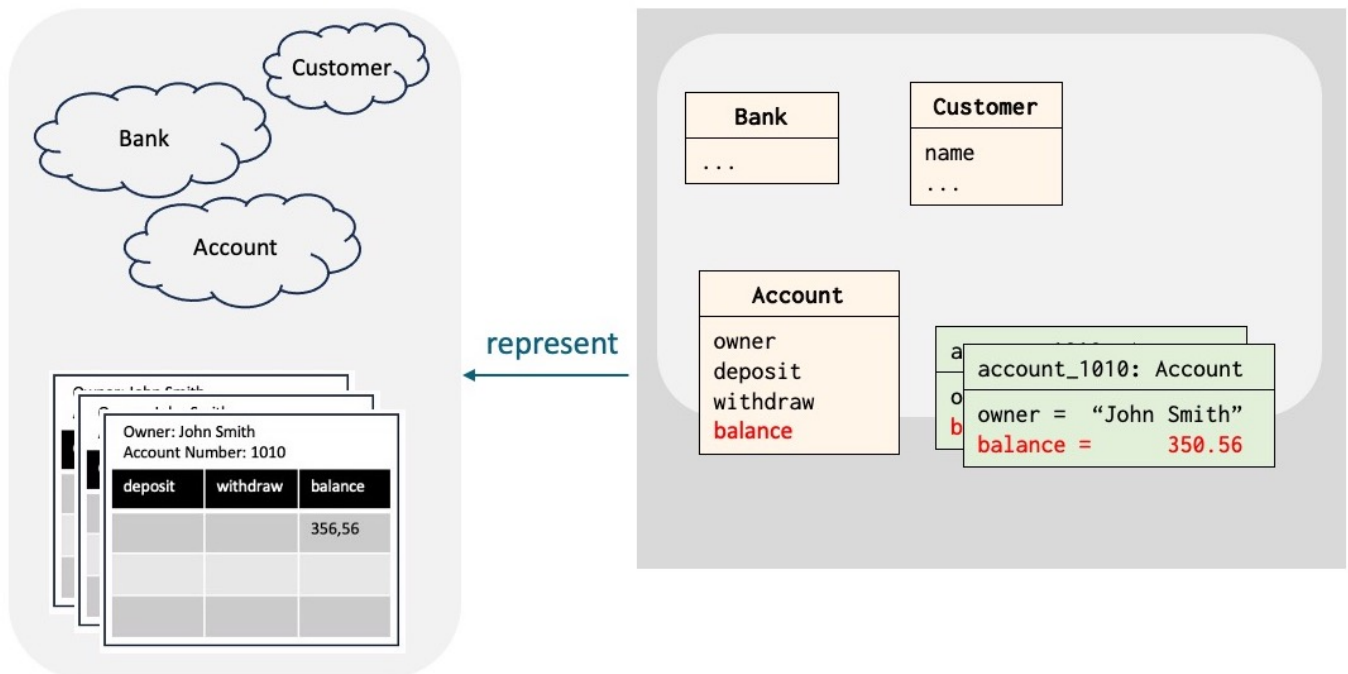
There will, however be elements in a given program that do not relate to phenomena or concepts from the application domain. This is due to the fact that it must be possible to execute the program on a computer. To do this, a program will have elements that implement the elements that relate to the model of the application domain.

The part of the program that describes phenomena and concepts from the application domain is called the *representative part*. The part which is not representative is called *non-representative*.



Non-representative parts of class `Account`

In the following we consider class `Account` in the bank domain with respect to representative and non-representative parts.



Representation of concepts and phenomena in the bank domain

In the previous versions of `Account` there is at a first glance no attributes that cannot be considered representative. All of `owner`, `balance`, `interestRate`, `deposit`, `withdraw`, etc. are meaningful in the bank domain.

We originally introduced assignments like:

```
account_1010.balance := 523.07
```

and said that such an assignment may be considered a deposit on the account. And further assignments like:

```
account_1010.balance := account_1010.balance + 200
account_1010.balance := account_1010.balance - 300
```

may also be considered deposits or withdrawals.

However, we have methods `deposit` and `withdraw`, which are supposed to represent deposits and withdrawals of money on the account. And from a modeling point of view these conflict with assigning to `balance` directly.

In addition, there is more to a `deposit` and `withdraw` than just updating `balance`. In section , we introduced class `Transactions` and an object `transaction`, which keeps track of deposits and withdrawals on a given account. We would thus want to prevent access to `balance` from outside the `Account`-objects.

However, we do want to be able to read/print the current balance of a given account. In addition, we may want to log that someone has read the balance of an account.

We may handle this by adding a method `getBalance` to the class `Account`:

```
getBalance -> bal: var float: transact
   bal := balance
   theTransaction.what := "getBalance"
```

The above considerations also apply to the variable `interestRate`. Here we do not want to be able to manipulate `interestRate` directly from outside an account, as update of an `interesRate` may imply more actions to be carried out. This may include informing the customer about the change.

We may thus introduce a method `getInterestRate` like `getBalance` and we may introduce methods `setInterestRate` and `getInterestRate`.

In the next section we introduce language mechanisms for controlling the accessibility of attributes.